Analysis of the impact of multi-threading on communication performance

François Trahay Élisabeth Brunet
Alexandre Denis

INRIA, LABRI, Université Bordeaux 1
351 cours de la Libération
F-33405 TALENCE, FRANCE
{trahay,brunet,denis}@labri.fr

Abstract

Although processor become massively multicore and by the way, applications multithreaded, the issue of multithreading in the current communication libraries is still neglected. Most of them content themselves with supporting concurrent flows thanks to coarse-grain locking but do not manage them in a parallel. In this paper, we firstly present how to refine the locking feature in order to perform simultaneaous operations. Secondly, we propose to multithread communication libraries themselves in order to offload operations on idle cores. Throughout the paper, we highlight the global benefits of those features on the communications.

1 Introduction

The current trend in cluster's architecture leads toward an increase of the number of cores per node. It becomes common to have 8 or 16 cores per machine and the evolution of processors will lead to the use of tens or hundreds of cores per node. Thus, the approach to exploit clusters has to evolve since the classical pure MPI model suffers from scalability limitations: the increase of the number of MPI processes per node may restrict the memory or TLB space.

In order to override these limitations, hybrid solutions that mix the use of threads and MPI processes seem to be the best candidate. Such paradigms allow to pool the hardware resources and to exploit them as much as possible. However the use of threads by applications requires some precaution in communication libraries in order to avoid race conditions when threads access concurrently the library.

- contribution of the paper: thread-safety can be done in many ways and has a cost. We analyze the benefit and cost of various solutions.

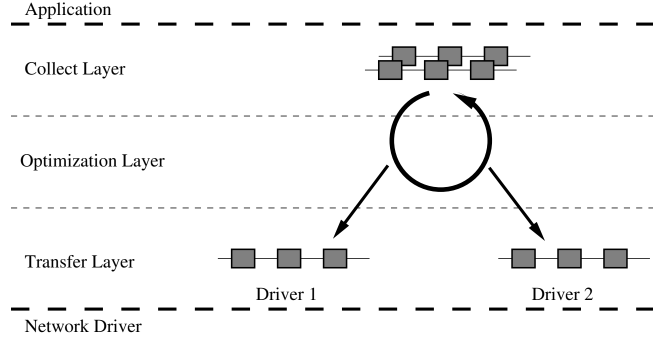

Figure 1. Architecture of NEWMADELEINE

1.1 Related work

Although processor development is clearly heading to a massive use of multicore processor, the issue of multithreading in communication library has received little attention in the literature. Most MPI implementations do not fully support the use of threads. However, MiMPI [5] is thread-safe and is able to use internal threads to make communication progress. Though, this implementation is only available for TCP and performs badly for small messages. USFMPI [4] is an MPI-1.2 implementation for TCP and Myrinet GM that supports multithreading. It also uses threads to make rendezvous handshakes progress in the background. MPICH-Madeleine [2] is a thread-safe MPI implementation that uses internal threads to make asynchronous communication progress. In [3], Balaji et al. present several approaches to build a thread-safe MPI implementation. These techniques are implemented in MPICH2, making it fully thread-safe. OpenMPI [6] provides the MPI_THREAD_MULTIPLE thread-safety level and can use a progression thread for TCP but these options are only lightly tested and some bugs remain.

1.2 Context of study

In order to analyze the impact of multi-threading on communications, we have conducted experiments with our PM2 software suite, composed of a communication library (NEWMADELEINE), a multithreading library (MARCEL), and an I/O event manager (PIOMAN).

Our communication library for high performance networks is called NEWMADELEINE [1] and is available over MX/Myrinet, Verbs/InfiniBand, Elan/QsNet, and TCP/Ethernet. As depicted in Figure 1, NEWMADELEINE has a 3-layer architecture with its activity driven by the underlying NICs, in contrast with the usual behavior of most communication libraries driven by the send/recv from the application. The core layer applies dynamic scheduling optimizations on multiple communication flows such as reordering, packet coalescing, multirail distribution, etc.

We used the MARCEL [7] multi-threading library. It features a two-level thread scheduler that achieves the performance of a user-level thread package while being able to exploit SMP machines.

The communication engine of the PM2 software suite is PIOMAN [9] event detector. It aims at providing the other software components with a service that guarantees a predefined level of reactivity to I/O events. PIOMAN is able to balance the event detection processing over the whole machine and thus works closely with the MARCEL thread scheduler which provides information on the running threads and the available CPUs. This way, PIOMAN is able to choose the most appropriate method (polling or

interrupt-based blocking call) depending on the context (number of computing threads, available CPUs, etc.) to ensure a high level of reactivity. PIOMAN uses the tasklet mechanism provided by MARCEL to execute detection methods on the most suitable CPUs. MARCEL also schedules PIOMAN on some triggers (CPU idleness, context switches, timer interrupts, etc.) so as to ensure a fast detection of communication events. PIOMAN is generic and is not bound to any particular network or communication library; however, in this paper we focus on the use of PIOMAN by NEWMADELEINE.

- thread safety v.s. multi-threaded communication engine
- various level of threads support
- we decompose and analyze the cost/benefit of each part of threads support We have implemented various features and profiled the code.

The benchmarks in the following Sections have been performed on a set of quad-core 3.16 GHz Xeon X5460 boxes with 4 GB of main memory running Linux version 2.6.26. Nodes are interconnected through Myricom Myri-10G NICs (with the MX 1.2.7 driver) and ConnectX Infiniband DDR (MT25418, with the OFED 1.3.1 driver). Latency graphs with a single rail have been obtained on Myrinet MX.

2 Impact of thread safety

In this Section, we study the various mechanisms required when designing a fully thread-safe communication library. We also evaluate the benefits and performance of such techniques.

2.1 Coarse-grain locking

The easiest way to protect a communication library from concurrent access consists in using a coarse-grain locking mechanism: each time a thread accesses the library, a mutex is held (see Figure 2). When the thread returns from the library, the mutex is released. Of course, the mutex is also released before entering a blocking section in order to avoid deadlocks. This method permits to support concurrent access without paying an expensive overhead: each access to the library implies only one locking operation.

As the mutex is held for a short period (a few microseconds at most), we use *spinlocks* to implement this coarse-grain locking mechanism within NEWMADELEINE. Thus, if a critical section is locked when a thread tries to access it, the thread waits actively until the mutex is released. In order to evaluate the overhead of this locking mechanism, we performed a pingpong test. The results we obtained are depicted on Figure 3. The use of a library-wide mutex (“coarse-grain locking” on the Figure) implies an overhead of 140 ns as the spinlock is held and released twice (once for submitting the message to the collect layer, once to transmit it through the network)

The global mutex permits to ensure thread-safety with a limited impact on latency but suffers from a lack of parallelism: as soon as a thread enters the library, the communication actions (polling, message submission, etc.) are limited to the one performed by this thread. Thus communication processing is serialized when several threads communicate. On Figure 4, we can see the results we obtained for a concurrent pingpong test: two threads perform pingpong tests concurrently. The results show that each thread’s latency roughtly corresponds to twice the normal latency. This is due to the mutual exclusion between the two threads that have to wait each other to access the communication library.

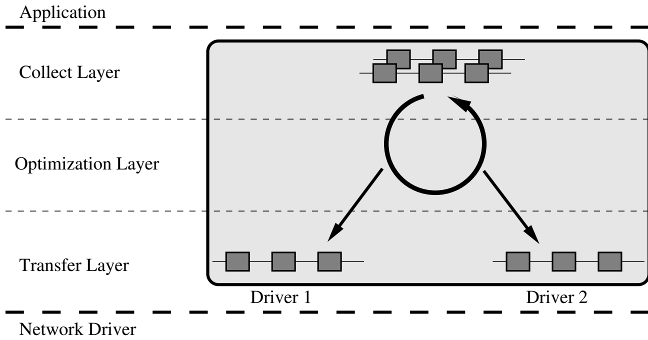

Figure 2. Coarse-grain locking in NEWMADELEINE

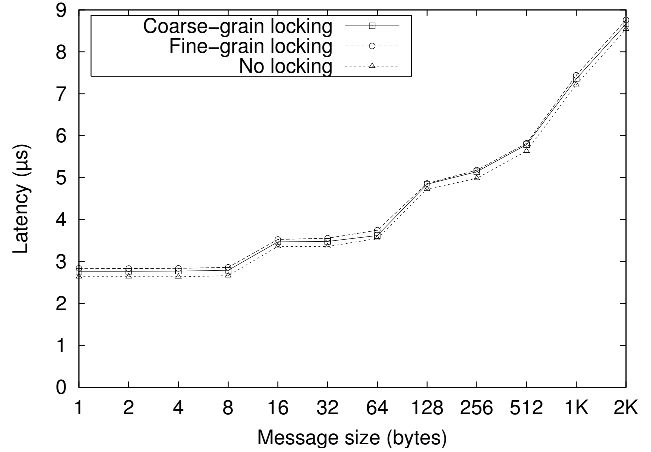

Figure 3. Impact of locking on latency over Myrinet MX

2.2 Fine-grain locking

In order to get an efficient thread-safe communication library, it is necessary to allow threads to process communication flows in parallel. Similar actions should still be performed under mutual exclusion (for instance, polling a "thread-unsafe" network) but unrelated processes should be parallelizible: it should be possible to send a message over a network while receiving data from another network. Polling in parallel over different networks should also be permitted.

To allow such parallel actions, locking has to get finer and critical sections has to be identified more precisely. In Section 1.2, we showed that global structures within NEWMADELEINE were limited to two kinds of lists:

- The packets to optimize in the collect layer. This list can be accessed by the application (through `nm_isend`, etc.) and by the optimization layer.
- The packets to send through the network in the transfer layer. These lists are accessed by the optimization layer and the transfer layer.

Implementing a fine-grain locking mechanism thus boils down to use separate mutexes for each list as it is shown on Figure 5. The lists of packets to optimize in the collect layer use a global mutex as the packet scheduler needs to iterate over those lists to generate a packet to send.

To evaluate the overhead of such mechanism, we performed a classical pingpong test. The results we obtained are shown on Figure 3. It appears that using a fine-grain locking introduces an overhead of 230 ns. This overhead is higher than the coarse-grain locking one (140 ns) because the number of mutexes is higher in the critical path.

Figure 4 shows the results we obtained for the concurrent pingpong test described in Section 2.1. Fine-grain locking performs better than coarse-grain locking as the communication library can be accessed simultaneously by several threads. The latency obtained by each thread is though higher than the latency obtained when using only one thread. This can be explained by the more intensive use the NIC and by the contention when accessing the different locks.

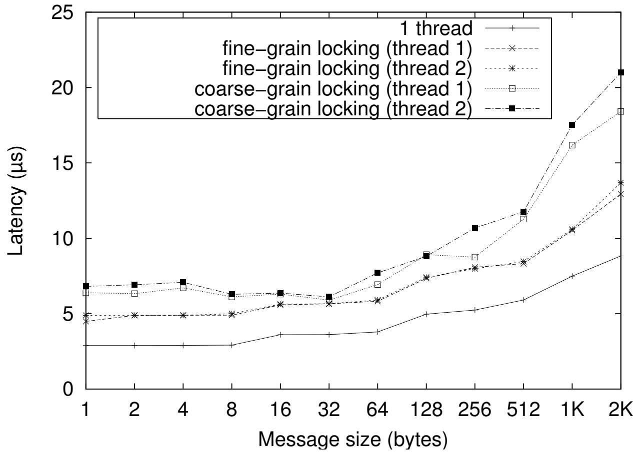

Figure 4. Two threads perform concurrently pingpong programs

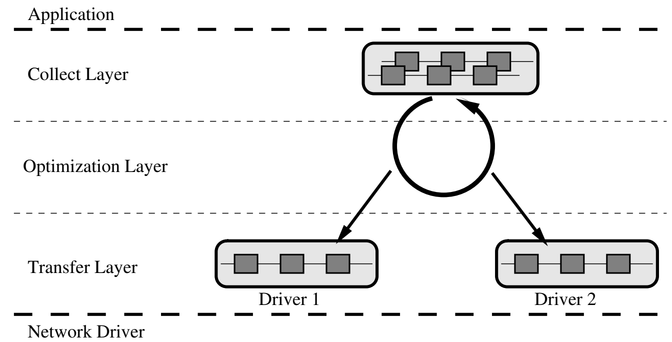

Figure 5. Fine grain locking in NEWMADELEINE

2.3 Busy waiting v.s. passive waiting

In order to design an efficient thread-safe communication library, it is necessary to understand its behavior in a multithreaded context. A classical problem when mixing threads and communication is the implementation of waiting functions (`MPI_Wait` for instance). Most regular (*i.e.* not thread-safe) communication library implement waiting function as busy waiting: when a thread waits for the end of a communication, it keeps polling until the corresponding network request succeeds.

Although this behavior is extremely efficient in a single-threaded environment, it can be problematic when multiple threads perform this operation in parallel. In this latter case, polling is done concurrently and thus contention may decrease each thread's polling frequency. Another problem here is that two or more CPUs perform a single operation (*i.e.* polling a network) that could be performed by one CPU. This would leave the others CPUs available to schedule application's threads.

In the thread scheduler's point of view, waiting threads should wait on blocking primitives (semaphores, conditions, etc.) in order to let other threads be scheduled. But if waiting functions are based on classical blocking primitives, no polling will be performed while a thread is blocked. The blocking primitives thus have to be modified to perform polling efficiently. In NEWMADELEINE, this is implemented by the PIOMAN progression engine that is called from the thread scheduler when a thread is about to block on a semaphore. This optimization requires modifications of the thread scheduler in order to add a few hooks at key points (CPU idleness, context switches, timer interrupts, etc.). These hooks are used to call PIOMAN and thus to poll the networks.

To evaluate the impact of using PIOMAN to poll the network, we performed the same latency test as in Section 2.1. The results depicted on Figure 6 show an overhead of 200 ns due to the management of internal lists as well as PIOMAN locking.

The use of blocking primitives also introduces an overhead since it implies expensive context switches. Figure 7 shows the results of the latency test when waiting functions are implemented with semaphores. It appears that the impact of the context switches on latency is rather high (750 ns).

It is thus important to avoid these context switches when possible. A solution consists in mixing active

and passive waiting: when a thread is about to block on a semaphore, it first polls for a short duration (for instance 5 μs) and then enters the semaphore. By doing this, the context switch is avoided if the wanted event occurs within 5 μs. If this event happens later, the context switch has to be paid but it represents a small percentage of the communication cost.

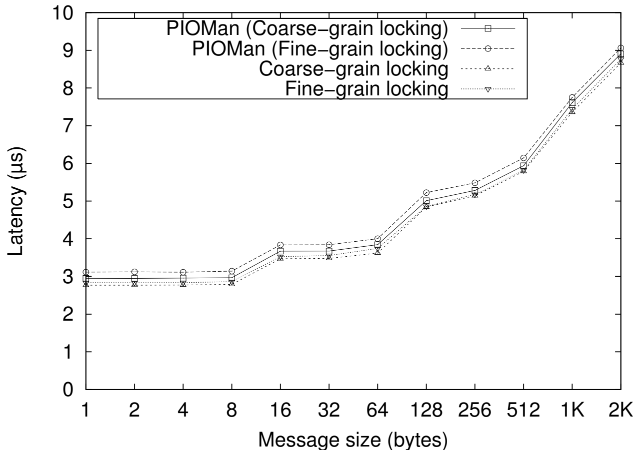

Figure 6. Impact of PIOMan on latency

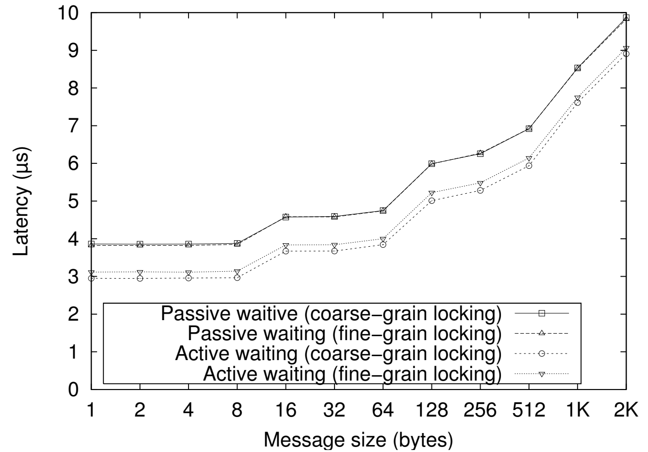

Figure 7. Impact of semaphores on latency

3 Multi-threading the communication engine

In the previous Section, we saw that designing an efficient thread-safe communication library was achievable. But the development of multicore chips and the increase of the number of cores per node make it important to *take advantage* of multithreading instead of *supporting* it.

In this Section, we study the impact of using idle cores to process communication flows. As we showed that using fine-grain locking within a communication library has a limited overhead, we only considere this type of locking mechanism in this Section.

3.1 First step: using idle cores to make communication progress

The increase of the number of cores per node may lead to *"holes"* in the scheduling: as the number of thread increases, the need for both intra-node and inter-node synchronization makes threads wait for incoming messages or mutex release. These holes can be used to make communication progress in the background. We showed that *rendezvous* handshakes can be detected on idle cores, allowing to overlap computation and communication of large messages [9].

This technique requires to poll the network on a core while the application computes on another core. To evaluate the overhead of defering the polling to a separate core, we performed a pingpong test that binds the main thread to a CPU.

The results we obtained for this test are depicted in Figure 8. The application thread is bound to CPU 0, thus polling on this CPU leads to better performance. Polling on a CPU that shares a L2-cache (CPU 1) implies an overhead of 400 ns and polling on a CPU that do not share a cache with CPU 0 (CPU 2 or CPU 3) leads to an overhead of 1.2 μs. These extra costs are due to communication between cores and cache misses.

The same test has been carried out on a dual quad-core Xeon machine and the results are similar: polling on a CPU that shares a cache (CPU 1) costs 400 ns, polling on the same chip but on a separate cache (CPU 2 or CPU 3) costs 2.3 μs and polling on another chip (CPU 4 to CPU 7) costs 3.1 μs.

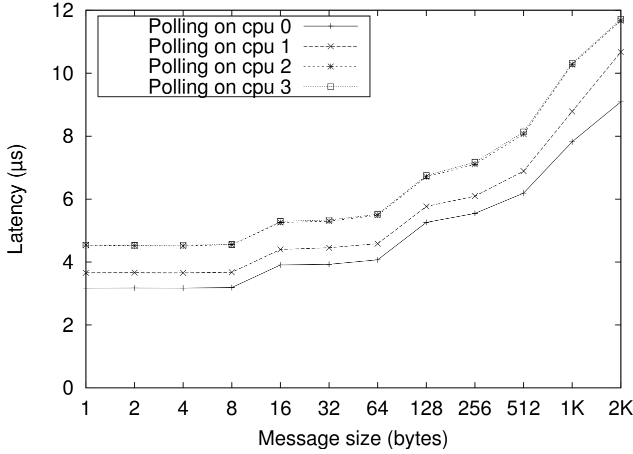

Figure 8. Impact of cache affinity on a quad-core chip

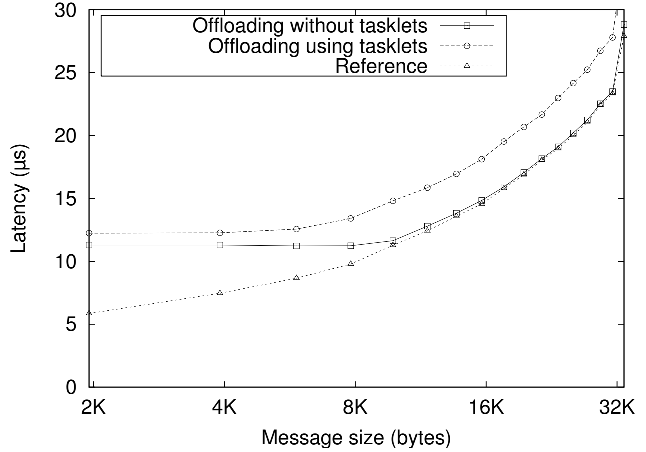

Figure 9. Impact of tasklets on defered message submission

3.2 Second step: using idle cores to perform cpu-intensive tasks

In addition to the progression of communication in the background, idle cores can be exploited to perform time-consuming operations such as message submission to the network [8]. This way, communication of small messages and computation are overlapped. In previous papers, PIOMAN relied extensively on tasklets [10] to offload communication processing. Tasklets are well-suited for such mechanisms as it offers a convenient way to defer a treatment.

It is though possible to offload communication processing without using tasklets: while a core is idle, MARCEL invokes pioman that can “detect” if a message needs to be submitted to a network. This requires some precautions since mutual exclusion has to be managed by hand.

In order to evaluate these two techniques of message submission offloading, we performed an overlapping test. The results depicted on Figure 9 show that offloading message submission with tasklet introduces an overhead of 2 μs whereas using idle cores to transmit the data (without any tasklet) costs 400 ns. This latter overhead corresponds to the cost of defering polling on idle core as explained in Section 3.1. The overhead of tasklets seems to be due to the complex locking mechanism involved when a tasklet is invoked.

4 Conlusion and future work

We have studied the impact of thread-safety when implementing a generic communication library. Even though the identification of critical sections can be tedious, very few modifications need to be applied to the implementation in order to efficiently support multithreading. Locking mechanisms re-

quired to do this have a limited impact on raw performance and permits to achieve good results when the application uses threads.

The impact of multithreading within the communication library has also been studied and it appears that using idle cores to process communication flows requires to take data locality and cache effects into account. Despite the convenience of tasklets to defer communication treatments to idle cores, such mechanism must be used carefully as it implies a significant overhead.

This contribution opens the path to future works such as using internal multithreading in the ongoing port of MPICH2-Nemesis over the NEWMADELEINE-PIOMAN software stack. This will allow us to benchmark our multithreaded communication library with real applications that mix multithreading and message passing.

Acknowledgements. This work has been funded by the project "LEGO" (ANR-CICG05-11) from the French National Agency for Research (ANR).

References

[1] O. Aumage, E. Brunet, N. Furmento, and R. Namyst. Newmadeleine: a fast communication scheduling engine for high performance networks. In *CAC 2007: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007*, March.

[2] O. Aumage, G. Mercier, and R. Namyst. MPICH-Madeleine: a True Multi-Protocol MPI for High-Performance Networks. In *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, page 51, San Francisco, Apr. 2001. IEEE.

[3] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward Efficient Support for Multithreaded MPI Communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 15th European Pvm/Mpi Users' Group Meeting, Dublin, Ireland, September 7-10, 2008, Proceedings*, page 120. Springer, 2008.

[4] S. Caglar, G. Benson, Q. Huang, and C. Chu. USFMPI: A Multi-threaded Implementation of MPI for Linux Clusters. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2003.

[5] F. Garcia, A. Calderón, and J. Carretero. Mimpi: A multithred-safe implementation of mpi. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 207–214, London, UK, 1999. Springer-Verlag.

[6] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A Flexible High Performance MPI. In *The 6th Annual International Conference on Parallel Processing and Applied Mathematics*, 2005.

[7] Runtime Team, LaBRI-Inria Bordeaux — Sud-Ouest. Marcel: A POSIX-compliant thread library for hierarchical multiprocessor machines, 2007. `http://runtime.bordeaux.inria.fr/marcel/`.

[8] F. Trahay, E. Brunet, A. Denis, and R. Namyst. A multithreaded communication engine for multicore architectures. In *CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, Apr. 2008. IEEE.

[9] F. Trahay, A. Denis, O. Aumage, and R. Namyst. Improving reactivity and communication overlap in mpi using a generic i/o manager. In *EuroPVM/MPI*. Springer, 2007.

[10] M. Wilcox. I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers. In *Linux.conf.au*, Perth, Australia, January 2003. The University of Western Australia.